LAMPIRAN

Lampiran 1. Kode Program Pemrosesan Tunggal pada Program Permainan Catur (single.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <mpi.h>
#include <sys/time.h>

int node_count, node_rank, node_namelen;
char node_name[MPI_MAX_PROCESSOR_NAME];
struct timeval time_start, time_stop;

// PIECE_VALUE
#define V_NULL              0           //value of null space
#define V_PAWN              100         //value of pawn
#define V_KNIGHT            300         //value of knight
#define V_BISHOP            325         //value of bishop
#define V_ROOK              500         //value of rook
#define V_QUEEN             900         //value of queen
#define V_KING              30000       //value of king
#define V_MAX               900000      //bestValue of minimax

// PIECE_CODE
#define EMPTY               0
#define PAWN                1
#define KNIGHT              2
#define BISHOP              3
#define ROOK                4
#define QUEEN               5
#define KING                6

#define MASK_COLOR          8       //1000
#define MASK_PIECE          7       //0111

#define COLOR_WHITE         0       //0000
#define COLOR_BLACK         8       //1000

#define A1  0
#define B1  1
#define C1  2
#define D1  3
#define E1  4
#define F1  5
#define G1  6
#define H1  7

/*
        piece = COLOR_X or PIECE_CODE
        1  = white pawn
        9  = black pawn
        10 = black knight
        dst
*/

//array 0 for dummy
int ply;
int depthply;
int init_board[64]={
        4, 2, 3, 5, 6, 3, 2, 4,
        1, 1, 1, 1, 1, 1, 1, 1,
```

```
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
     9, 9, 9, 9, 9, 9, 9, 9,
     12,10,11,13,14,11,10,12
    };

int false_init_board[64]={
        4, 2, 0, 0, 6, 0, 0, 4,
        0, 1, 1, 1, 1, 1, 1, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 2, 0, 0, 5,
        0, 0, 0, 0, 0, 0, 0, 0,
    1, 9, 9, 9, 9, 9, 9, 1,
        0, 0, 0,13,14,11, 0, 0
    };

int  board[64];
int tboard[64];
int history_board[64][64];


bool slide[7] = {
        false, false, false, true, true, true, false
};

int number_of_move[7] = {
        0, 0, 8, 4, 4, 8, 8
};

int pieces_value[7] = {
        V_NULL,
        V_PAWN,
        V_KNIGHT,
        V_BISHOP,
        V_ROOK,
        V_QUEEN,
        V_KING
        };

int movement[7][8] = {
        {   0,   0,   0,   0,  0,  0,  0,  0,  0 },      //dummy
        {   10,  9,  11,   0,-10, -9,-11,  0 },      //pawn
        { -21, -19, -12, -8,  8, 12, 19, 21 },      //dullahan knight
        { -11,  -9,   9, 11,  0,  0,  0,  0 },      //bishop
        { -10,  -1,   1, 10,  0,  0,  0,  0 },      //barathum rook
        { -11, -10,  -9, -1,  1,  9, 10, 11 },      //queen of fire, candidate for
stepmother
        { -11, -10,  -9, -1,  1,  9, 10, 11 }       //useless king
};

int mailbox[120] = {
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
         -1,  0,  1,  2,  3,  4,  5,  6,  7, -1,
         -1,  8,  9, 10, 11, 12, 13, 14, 15, -1,
         -1, 16, 17, 18, 19, 20, 21, 22, 23, -1,
         -1, 24, 25, 26, 27, 28, 29, 30, 31, -1,
         -1, 32, 33, 34, 35, 36, 37, 38, 39, -1,
         -1, 40, 41, 42, 43, 44, 45, 46, 47, -1,
         -1, 48, 49, 50, 51, 52, 53, 54, 55, -1,
         -1, 56, 57, 58, 59, 60, 61, 62, 63, -1,
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
```

```
};

int mailbox64[64] = {
        21, 22, 23, 24, 25, 26, 27, 28,
        31, 32, 33, 34, 35, 36, 37, 38,
        41, 42, 43, 44, 45, 46, 47, 48,
        51, 52, 53, 54, 55, 56, 57, 58,
        61, 62, 63, 64, 65, 66, 67, 68,
        71, 72, 73, 74, 75, 76, 77, 78,
        81, 82, 83, 84, 85, 86, 87, 88,
        91, 92, 93, 94, 95, 96, 97, 98
};

int castling_white_left[2]  = {  5,  6};
int castling_white_right[3] = {  1,  2, 3};
int castling_black_left[3]  = { 57, 58, 59};
int castling_black_right[2] = { 61, 62};

bool bWhiteAllowCastlingL;
bool bWhiteAllowCastlingR;
bool bBlackAllowCastlingL;
bool bBlackAllowCastlingR;

//bool bWhiteCheck=false;
//bool bBlackCheck=false;

int start_pos_move;
int des_pos_move;

//FILE* fftest;

int evaluate();
int evaluatebd(int bd[64]);
int maxi( int depth, bool maximize );
bool check_move(int piece_pos, int des_move);
bool check_target_color(int piece_pos, int des_move);
bool check_pawn_move(int piece_pos, int des_move);
bool check_move_player(int piece_pos, int des_move);
int move_player_piece(int piece_pos, int des_move);
int move_pseudo_piece(int piece_pos, int des_move);
int move_pseudo_pawn(int piece_pos, int des_move);
int move_piece(int piece_pos, int des_move);

void copy_board(int board_des[64], int board_src[64]);
void display(int bd[64]);
void printboard(int bd[64],FILE* tfile);
int not_main();

int evaluate()
{
        int i;
        int total=0;
        int lpiece;
        int lcolor;

        for(i=0;i<64;i++)
        {

                lpiece = tboard[i] & MASK_PIECE;
                lcolor = tboard[i] & MASK_COLOR;
                if(lcolor == COLOR_WHITE)
                {
                        total = total + pieces_value[lpiece];
                }else
                {
                        total = total - pieces_value[lpiece];
```

```
                }
        }
        return total;
}

int evaluatebd(int bd[64])
{
        int i;
        int total=0;
        int lpiece;
        int lcolor;

        for(i=0;i<64;i++)
        {

                lpiece = bd[i] & MASK_PIECE;
                lcolor = bd[i] & MASK_COLOR;
                if(lcolor == COLOR_WHITE)
                {
                        total = total + pieces_value[lpiece];
                }else
                {
                        total = total - pieces_value[lpiece];
                }
        }
        return total;
}

/*
        this is our monster!!
*/
int maxi( int depth, bool maximize ) {
        int max;                //max or min
        int i;                  //scan board
        int n_move;             //number of move
        int score;              //score for minimax
        int lpiece;             //piece identification
        int lcolor;             //color of piece
        int des_move;           //destination move
        int lbestpos;
        int lbestmove;

    if ( depth <= 0 ) {
                return evaluate();
                }

        if ( maximize )
        {
            //printf("depth %d\n ",depth);
         max = -V_MAX;

                copy_board( history_board[depth], tboard);
                //record history (of three kingdom)
                for ( i=0; i<64; i++ ) {            //scan board
                        lpiece = tboard[i] & MASK_PIECE;
                        lcolor = tboard[i] & MASK_COLOR;
                        if(( lpiece != EMPTY) && ( lcolor == COLOR_WHITE))
                        {
                                if(lpiece==PAWN)
                                {
                                        for ( n_move = 0; n_move < 3; n_move++ )
                                        {
                                                des_move = movement[lpiece][n_move];
                                /* des_move = -movement[lpiece][n_move]; if its BLACK */
                                                if(check_pawn_move(i,des_move))
                                                {
```

```
                                        move_pseudo_pawn(i,des_move);
                                        score = maxi( depth - 1, false );
                                        if(score>max)
                                                {
                                                        max=score;

                                                }
                                        copy_board(tboard,
history_board[depth]);
                                }
                        }

                }else
                {
                        for ( n_move = 0; n_move <
number_of_move[lpiece]; n_move++ )
                        {
                                des_move=0;
                                do
                                {
        des_move=des_move+movement[lpiece][n_move];
                                        if(!check_move(i,des_move))
                                        {
                                                break;
                                        }

                                        if(check_target_color(i,des_move))
        //if capture
                                        {

        move_pseudo_piece(i,des_move);
                                                score = maxi( depth - 1,
false );
                                                if(score>max)
                                                {
                                                        max=score;


                                                }
                                                copy_board(tboard,
history_board[depth]);        //return board
                                                break; //stop
                                        }
                                //if empty
                                        move_pseudo_piece(i,des_move);
                                        score = maxi( depth - 1, false );
                                        if(score>max)
                                                {
                                                        max=score;

                                                }
                                        copy_board(tboard,
history_board[depth]);              //return board

                                }while(slide[lpiece]);

                        }
                        //castling move white
                        /*

        if((lpiece==KING)&&(bWhiteAllowCastlingL)&&(bWhiteAllowCastlingR))
                        {
                                for()
                                {

                                }
```

```
                                }
                                */
                        }
                }
        }

        //printf("\nstart pos: %d destination: %d\n
",start_pos_move,des_pos_move);
      //printf("value %d \n ",max);
        return max;
      }else   //minimum
      {
          //printf("depth %d\n ",depth);
       max = V_MAX;

        copy_board( history_board[depth], tboard);
        //record history (of three kingdom)
        for ( i=0; i<64; i++ ) {                    //scan board
                lpiece = tboard[i] & MASK_PIECE;
                lcolor = tboard[i] & MASK_COLOR;
                if(( lpiece != EMPTY) && ( lcolor == COLOR_BLACK))
                {
                        if(lpiece==PAWN)
                        {
                                for ( n_move = 0; n_move < 3; n_move++ )
                                {
                                        des_move = -movement[lpiece][n_move];
                                /* des_move = -movement[lpiece][n_move]; if its BLACK */
                                        if(check_pawn_move(i,des_move))
                                        {
                                                move_pseudo_pawn(i,des_move);
                                                score = maxi( depth - 1, true );
                                                if(score<max)
                                                        {
                                                                max=score;
                                                                lbestpos=i;
                                                                lbestmove=des_move;
                                                        }
                                                copy_board(tboard,
history_board[depth]);
                                        }
                                }

                        }else
                        {
                                for ( n_move = 0; n_move <
number_of_move[lpiece]; n_move++ )
                                {
                                        des_move=0;
                                        do
                                        {

      des_move=des_move+movement[lpiece][n_move];
                                                if(!check_move(i,des_move))
                                                {
                                                        break;
                                                }

                                                if(check_target_color(i,des_move))

      //if capture
                                                {

      move_pseudo_piece(i,des_move);
                                                        score = maxi( depth - 1,
true );
                                                        if(score<max)
```

```
                                        {
                                                max=score;
                                                lbestpos=i;
                                                lbestmove=des_move;
                                        }
                                        copy_board(tboard,
history_board[depth]);        //return board
                                        break; //stop
                                }

                                move_pseudo_piece(i,des_move);
                                score = maxi( depth - 1, true );
                                if(score<max)
                                        {
                                                max=score;
                                                lbestpos=i;
                                                lbestmove=des_move;
                                        }
                                copy_board(tboard,
history_board[depth]);                //return board


                                        }while(slide[lpiece]);

                                }
                                //castling move white
                                /*

        if((lpiece==KING)&&(bWhiteAllowCastlingL)&&(bWhiteAllowCastlingR))
                                {
                                        for()
                                        {

                                        }
                                }
                                */
                        }
                }
        }

         start_pos_move=lbestpos;
         des_pos_move=lbestmove;
                //printf("\nstart pos: %d destination: %d\n
",start_pos_move,des_pos_move);
         //printf("value %d \n ",max);

                return max;
        }
}
 /**
        check_move mechanism:
        1. check if out of bound
        2. check if empty
        3. check friendly piece

 */
bool check_move(int piece_pos, int des_move)
{
        int lpiece=board[piece_pos] & MASK_PIECE;
        int lcolor=board[piece_pos] & MASK_COLOR;
        int lcolor_target;
        int lpiece_target;

        int mb_pos=mailbox64[piece_pos];            //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index
```

```
	//1. check boundary
	if(destination==-1)
	{
		//out of bound
		return false;
	}else{
		lcolor_target = tboard[destination] & MASK_COLOR;
		lpiece_target = tboard[destination] & MASK_PIECE;
		if((lpiece_target) == EMPTY) //2. check if empty
		{
			return true;
		}else				//3. if not empty, check their color
		{

			if(lcolor==lcolor_target)	//you cannot attack same color
			{
				//hold your attack, its our ally
				return false;
			}else
			{
				//yes, kill him!!
				return true;
			}
		}
	}
}

//check if its empty and have different color, return true if they have different
color
bool check_target_color(int piece_pos, int des_move)
{
	int lpiece=tboard[piece_pos] & MASK_PIECE;
	int lcolor=tboard[piece_pos] & MASK_COLOR;
	int lpiece_target;
	int lcolor_target;

	int mb_pos=mailbox64[piece_pos];		//convert board into mailbox
	int destination = mailbox[mb_pos+des_move];
	//if destination = -1 its out of bound, otherwise board array index

	if(destination==-1)
	{
		//out of bound,return -1
		return false;
	}else
	{
		lpiece_target=tboard[destination] & MASK_PIECE;
		lcolor_target=tboard[destination] & MASK_COLOR;
		if(lpiece_target==EMPTY)
		{
			return false;
		}else
		{
			if(lcolor==lcolor_target)
			{
				return false;
			}else
			{
				return true;
			}
		}
	}
}
```

```
bool check_pawn_move(int piece_pos, int des_move)
{
	int lpiece=board[piece_pos] & MASK_PIECE;
	int lcolor=board[piece_pos] & MASK_COLOR;
	int lcolor_target;
	int lpiece_target;

	int mb_pos=mailbox64[piece_pos];		//convert board into mailbox
	int destination = mailbox[mb_pos+des_move];
	//if destination = -1 its out of bound, otherwise board array index

	//1. check boundary
	if(destination==-1)
	{
		//out of bound
		return false;
	}else{
		lcolor_target=tboard[destination] & MASK_COLOR;
		lpiece_target=tboard[destination] & MASK_PIECE;
		if((des_move==10)||(des_move==-10))
		{
			if(lpiece_target == 0)		//2. check if empty
			{
				return true;
			}else
			{
				return false;
			}
		}
		if((des_move==9)||(des_move==-9)||(des_move==11)||(des_move==-11))
		{
			if( (lpiece_target != 0) && (lcolor_target!=lcolor))
			//2. check if empty
			{
				return true;
			}else
			{
				return false;
			}
		}
	}
}

//exclusive for player
int move_player_piece(int piece_pos, int des_move)
{
    int piece_type=board[piece_pos]&MASK_PIECE;
    board[des_move]=board[piece_pos];
	board[piece_pos]=EMPTY;

	//promotion
	if(piece_type==PAWN)
    {
	 if(des_move>=56)
	 {
	     board[des_move]=QUEEN;
	 }
    }
    if(piece_type==ROOK)
    {
	 if(piece_pos==0)
	     {bWhiteAllowCastlingL=false;}
	 if(piece_pos==7)
	     {bWhiteAllowCastlingR=false;}
    }
    if(piece_type==KING)
```

```
    {
        bWhiteAllowCastlingL=false;
        bWhiteAllowCastlingR=false;
        //castling left
        if(des_move==C1)
        {
            board[A1]=EMPTY;
            board[C1]=KING;
            board[D1]=ROOK;
        }
        //castling right
        if(des_move==G1)
        {
            board[H1]=EMPTY;
            board[G1]=KING;
            board[F1]=ROOK;
        }
    }
    return 0;
}


bool check_move_player(int player_start, int player_des)
{
        int i;
        bool m_match = false;
        int lpiece = board[player_start] & MASK_PIECE;
        int lcolor = board[player_start] & MASK_COLOR;
        int lcolor_target;
        int lpiece_target;
        int mb_pos = mailbox64[player_start];      //convert board into mailbox
        int destination = 0;
        //if destination = -1 its out of bound, otherwise board array index
        int    startDesDiff = 0;
        int checkDesMove;
        int allowedMove[32];
        int allowedMoveCount=0;

        //check piece
        if(lpiece==EMPTY)
               { return false;}
        if(lcolor!=COLOR_WHITE)
               { return false;}

        //check movement rule
        if(lpiece==PAWN)
        {
         //only allow movement if player_des - player_start= 7/8/9
         startDesDiff=player_des-player_start;
         lpiece_target=board[player_des] & MASK_PIECE;
         lcolor_target=board[player_des] & MASK_COLOR;
         switch(startDesDiff)
         {
         case 8:
             if(lpiece_target==EMPTY)
             {
                 return true;
             }
             break;
         case 7:
             if(lcolor_target==COLOR_BLACK)
             {
                 return true;
             }else
             {
                 return false;
```

```
            }
            break;
        case 9:
            if(lcolor_target==COLOR_BLACK)
            {
                return true;
            }else
            {
                return false;
            }
            break;
        default:
            return false;
            break;
        };

    }
    else
{
    for(i=0;i<number_of_move[lpiece];i++)
    {
            checkDesMove=0;
            do
            {
                checkDesMove = checkDesMove+movement[lpiece][i];
                destination = mailbox[mb_pos+checkDesMove];
                //out of bound
                if(destination==-1)
                {
                    break;
                }

                lpiece_target=board[destination] & MASK_PIECE;
                lcolor_target=board[destination] & MASK_COLOR;

                //friendly fire
                if((lpiece_target!=EMPTY)&&(lcolor_target==COLOR_WHITE))
                {
                    break;
                }

                if((lpiece_target!=EMPTY)&&(lcolor_target==COLOR_BLACK))
                {
                    allowedMove[allowedMoveCount]=destination;
                    allowedMoveCount++;
                    break;
                }

                if(lpiece_target==EMPTY)
                {
                    allowedMove[allowedMoveCount]=destination;
                    allowedMoveCount++;
                }

            }while(slide[lpiece]);
    }
    if(allowedMoveCount>0)
    {
        for(i=0;i<allowedMoveCount;i++)
        {
            if(allowedMove[i]==player_des)
            {
                m_match=true;
            }
        }
    }
```

```
        //castling check
        if(lpiece==KING)
        {
            if((player_des==C1)&&(bWhiteAllowCastlingL)&&(board[A1]==ROOK))
            {
                if((board[B1]==EMPTY)&&(board[C1]==EMPTY)&&(board[D1]==EMPTY))
                {
                    return true;
                }
            }
            if((player_des==G1)&&(bWhiteAllowCastlingR)&&(board[H1]==ROOK))
            {
                if((board[F1]==EMPTY)&&(board[G1]==EMPTY))
                {
                    return true;
                }
            }
        }
    }
    return m_match;

}

//return destination array
int move_pseudo_piece(int piece_pos, int des_move)
{
        int tpiece=tboard[piece_pos];
        int mb_pos=mailbox64[piece_pos];          //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index


        //printf("\npos+move %d\ndestin %d\n ",mb_pos+des_move,destination);
        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
                tboard[destination]=tpiece;
                tboard[piece_pos]=EMPTY;

//         display(tboard);
//              printf("\nmove %d to %d\n ",piece_pos,destination);
//         printboard(tboard,fftest);

                return destination;
        }
}

//special move for pawn (including queen promotion)
int move_pseudo_pawn(int piece_pos, int des_move)
{
        int tpiece=tboard[piece_pos];
        int lcolor=board[piece_pos] & MASK_COLOR;
        int mb_pos=mailbox64[piece_pos];          //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index

        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
                tboard[piece_pos]=EMPTY;
                tboard[destination]=tpiece;
                if(lcolor==COLOR_BLACK)
```

```
                {
                        if(destination<8)
                        {
                                tboard[destination]=QUEEN+COLOR_BLACK;
                                //color white=0, so queen+color white=5+0 :P
                        }
                }
                if(lcolor==COLOR_WHITE)
                {
                        if(destination>55)
                        {
                                tboard[destination]=QUEEN+COLOR_WHITE;
                                //color white=0, so queen+color white=5+0 :P
                        }
                }
//      display(tboard);
//      printf("\nmove %d to %d\n ",piece_pos,destination);
//      printboard(tboard,fftest);
                return destination;
        }
}


//return destination array, its black move/AI move only
int move_piece(int piece_pos, int des_move)
{
        int tpiece=board[piece_pos];
        int piece_type=board[piece_pos] & MASK_PIECE;
        int mb_pos=mailbox64[piece_pos];            //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index

        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
                board[destination]=tpiece;
                board[piece_pos]=EMPTY;
                //promotion for black
                if((piece_type==PAWN)&&(destination<8))
         {
             board[destination]=COLOR_BLACK+QUEEN;
         }
                return destination;
        }
}

//special move for pawn (including queen promotion)
int move_pawn(int piece_pos, int des_move)
{
        int tpiece=board[piece_pos];
        int lcolor=board[piece_pos] & MASK_COLOR;
        int mb_pos=mailbox64[piece_pos];            //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index

        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
            board[destination]=tpiece;
                board[piece_pos]=EMPTY;
                if(lcolor==COLOR_BLACK)
                {
```

```
                        if(destination<8)
                        {
                                board[destination]=QUEEN+COLOR_BLACK;
                                //color white=0, so queen+color white=5+0 :P
                        }
                }
                return destination;
        }
}

int find_max(int value_a, int value_b)
{
        if(value_a>value_b)
        {
                return value_a;
        }else
        {
                return value_b;
        }
}

int find_min(int value_a, int value_b)
{
        if(value_a<=value_b)
        {
                return value_a;
        }else
        {
                return value_b;
        }
}

void copy_board(int board_des[64], int board_src[64])
{
        int i;
        for(i=0;i<64;i++)
        {
                board_des[i]=board_src[i];
        }
}
void display(int bd[64])
{
        int i;
        printf(" ");
        for(i=0;i<64;i++)
        {
            if(bd[i]==0)
            {
                printf(" .");
            }else
            {
                printf(" %c",bd[i]+'a');
            }
                if ((i + 1) % 8 == 0 && i != 63)
                        printf("\n ");
        }
        printf("\n");
}

void printboard(int bd[64],FILE* tfile)
{
        int i;
        fprintf(tfile," ");
        for(i=0;i<64;i++)
        {
            if(bd[i]==0)
```

```
            {
                fprintf(tfile," .");
            }else
            {
                fprintf(tfile," %c",bd[i]+'a');
            }
               if ((i + 1) % 8 == 0 && i != 63)
                     fprintf(tfile,"\n ");
        }
        fprintf(tfile,"\n");
        fprintf(tfile,"temporary value %d\n\n",evaluate());
}
int check_winning_player()
{
    int i;
    int winner=0;

    for(i=0;i<64;i++)
    {
        if(board[i]==KING)  winner++;
        if(board[i]==(COLOR_BLACK+KING))  winner--;
    }
    return winner;
}
int check_input(char* ch)
{
    int tpos=-1, tdes=-1;
    if((ch[0]<'a')||(ch[0]>'h')) return -1;
    if((ch[1]<'1')||(ch[1]>'8')) return -1;
    if((ch[2]<'a')||(ch[2]>'h')) return -1;
    if((ch[3]<'1')||(ch[3]>'8')) return -1;

    tpos=ch[0]-'a';
    tpos=tpos+(ch[1]-'1')*8;
    tdes=ch[2]-'a';
    tdes=tdes+(ch[3]-'1')*8;
    return (tpos<<6)|tdes;
}

//its just dumb main function
int main(int argc, char **argv)
{
        // MPI inits
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &node_count);
        MPI_Comm_rank(MPI_COMM_WORLD, &node_rank);
        MPI_Get_processor_name(node_name, &node_namelen);

    char c1[16],c2;
    int tpos,tdes;
    int input_return;
    int winner=0;
    bool gameover=false;
    //fftest=fopen("../filetest.txt","w");
    //if(fftest==NULL) return -1;

    bWhiteAllowCastlingL=true;
        bWhiteAllowCastlingR=true;
        copy_board(board, init_board);
        while(!gameover)
        {
                if(ply%2==0)//white one
                {
                        //input
                        //check
                        //if(check)
```

```
                    //-execute
                    //-ply++
                    display(board);
                    printf("\n input your move ");
                    scanf("%s",&c1);
                    input_return=check_input(c1);
                    if(input_return<0)
            {
                printf("\nwrong input, try again\n\n");
            }else
            {
                tpos=input_return>>6;
                tdes=input_return&63;
                printf("\nmove from %d to %d\n",tpos,tdes);
                if(check_move_player(tpos,tdes))
                {
                    move_player_piece(tpos,tdes);
                    ply++;
                }else
                {
                    printf("\nOops!! illegal move, try again\n\n");
                }
            }

              }else           //computer's turn
              {
                 if(ply<3)   //opening book
            {
                move_piece(52,-10);
            }else
            {
                copy_board(tboard,board);

                      gettimeofday(&time_start,NULL);
                      maxi(6,false);
                      gettimeofday(&time_stop,NULL);

        printf("elapsed= %.0f usec\n",(time_stop.tv_sec-time_start.tv_sec)*1e6+\
                                      (time_stop.tv_usec-time_start.tv_usec));
                      move_piece(start_pos_move, des_pos_move);
                //printf("\nbest move %d %d\n",start_pos_move,des_pos_move);

                display(board);
                //printf("\nvalue %d\n",evaluate());
            }
                      ply++;
              }
              winner=check_winning_player();
              if(winner!=0)
         {
            gameover=true;
            if(winner==1)
            {
                printf("\n\nwinner is WHITE\n");
            }else
            {
                printf("\n\nwinner is BLACK\n");
            }
         }
        }
        //fclose(fftest);
        MPI_Finalize();
        return(0);
    }

    /* //from wiki-wikipedia
```

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := -∞
        for each child of node
            val := minimax(child, depth - 1, FALSE)
            bestValue := max(bestValue, val)
        return bestValue
    else
        bestValue := +∞
        for each child of node
            val := minimax(child, depth - 1, TRUE)
            bestValue := min(bestValue, val)
        return bestValue
*/
```

Lampiran 2. Kode Program Pemrosesan Paralel pada Program Permainan Catur (multiple.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <mpi.h>
#include <sys/time.h>
#include <unistd.h>

// MPI tags
#define TAG_POS_STR 2
#define TAG_POS_DES 3
#define TAG_VAL 4
#define TAG_BOARD 5
#define TAG_PLY 6

//AI's thinking depth, must be even
#define DEPTH 2

// PIECE_VALUE
#define V_NULL              0           //value of null space
#define V_PAWN              100         //value of pawn
#define V_KNIGHT            300         //value of knight
#define V_BISHOP            325         //value of bishop
#define V_ROOK              500         //value of rook
#define V_QUEEN             900         //value of queen
#define V_KING              30000       //value of king
#define V_MAX               900000      //bestValue of minimax

// PIECE_CODE
#define EMPTY               0
#define PAWN                1
#define KNIGHT              2
#define BISHOP              3
#define ROOK                4
#define QUEEN               5
#define KING                6

#define MASK_COLOR          8       //1000
#define MASK_PIECE          7       //0111

#define COLOR_WHITE         0       //0000
#define COLOR_BLACK         8       //1000

#define A1  0
#define B1  1
#define C1  2
```

```
#define D1  3
#define E1  4
#define F1  5
#define G1  6
#define H1  7

/*
      piece = COLOR_X or PIECE_CODE
      1  = white pawn
      9  = black pawn
      10 = black knight
      dst
*/

// global vars
int node_count, node_rank, node_namelen;
char node_name[MPI_MAX_PROCESSOR_NAME];
struct timeval time_start, time_stop, time_start_global, time_stop_global;
MPI_Status status;
int i_start, i_stop;
int start_pos_move, des_pos_move, value_pos_move;
int *start_pos_move_global, *des_pos_move_global, *value_pos_move_global;

//array 0 for dummy
int ply;
int depthply;
int init_board[64]={
       4, 2, 3, 5, 6, 3, 2, 4,
       1, 1, 1, 1, 1, 1, 1, 1,
       0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0,
    9, 9, 9, 9, 9, 9, 9, 9,
    12,10,11,13,14,11,10,12
   };

int false_init_board[64]={
       4, 2, 0, 0, 6, 0, 0, 4,
       0, 1, 1, 1, 1, 1, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 2, 0, 0, 5,
       0, 0, 0, 0, 0, 0, 0, 0,
    1, 9, 9, 9, 9, 9, 9, 1,
       0, 0, 0,13,14,11, 0, 0
   };

int  board[64];
int tboard[64];
int history_board[64][64]; // of what kingdom?

bool slide[7] = {
       false, false, false, true, true, true, false
};

int number_of_move[7] = {
       0, 0, 8, 4, 4, 8, 8
};

int pieces_value[7] = {
       V_NULL,
       V_PAWN,
       V_KNIGHT,
       V_BISHOP,
       V_ROOK,
```

```
        V_QUEEN,
        V_KING
        };

int movement[7][8] = {
        {   0,   0,   0,  0,  0,  0,  0,  0 },      //dummy
        {   10,  9,  11,  0,-10, -9,-11,  0 },      //pawn
        { -21, -19, -12, -8,  8, 12, 19, 21 },      //dullahan knight
        { -11,  -9,   9, 11,  0,  0,  0,  0 },      //bishop
        { -10,  -1,   1, 10,  0,  0,  0,  0 },      //barathum rook
        { -11, -10,  -9, -1,  1,  9, 10, 11 },      //queen of fire, candidate for
stepmother
        { -11, -10,  -9, -1,  1,  9, 10, 11 }       //useless king
};

int mailbox[120] = {
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
         -1,  0,  1,  2,  3,  4,  5,  6,  7, -1,
         -1,  8,  9, 10, 11, 12, 13, 14, 15, -1,
         -1, 16, 17, 18, 19, 20, 21, 22, 23, -1,
         -1, 24, 25, 26, 27, 28, 29, 30, 31, -1,
         -1, 32, 33, 34, 35, 36, 37, 38, 39, -1,
         -1, 40, 41, 42, 43, 44, 45, 46, 47, -1,
         -1, 48, 49, 50, 51, 52, 53, 54, 55, -1,
         -1, 56, 57, 58, 59, 60, 61, 62, 63, -1,
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
         -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
};

int mailbox64[64] = {
        21, 22, 23, 24, 25, 26, 27, 28,
        31, 32, 33, 34, 35, 36, 37, 38,
        41, 42, 43, 44, 45, 46, 47, 48,
        51, 52, 53, 54, 55, 56, 57, 58,
        61, 62, 63, 64, 65, 66, 67, 68,
        71, 72, 73, 74, 75, 76, 77, 78,
        81, 82, 83, 84, 85, 86, 87, 88,
        91, 92, 93, 94, 95, 96, 97, 98
};

int castling_white_left[2]  = {  5,  6};
int castling_white_right[3] = {  1,  2, 3};
int castling_black_left[3]  = { 57, 58, 59};
int castling_black_right[2] = { 61, 62};

bool bWhiteAllowCastlingL;
bool bWhiteAllowCastlingR;
bool bBlackAllowCastlingL;
bool bBlackAllowCastlingR;
//bool bWhiteCheck=false;
//bool bBlackCheck=false;

//FILE* fftest;

int evaluate();
int evaluatebd(int bd[64]);
int maxi( int depth, bool maximize, int start, int stop );
bool check_move(int piece_pos, int des_move);
bool check_target_color(int piece_pos, int des_move);
bool check_pawn_move(int piece_pos, int des_move);
bool check_move_player(int piece_pos, int des_move);
int move_player_piece(int piece_pos, int des_move);
int move_pseudo_piece(int piece_pos, int des_move);
int move_pseudo_pawn(int piece_pos, int des_move);
int move_piece(int piece_pos, int des_move);
```

```
void copy_board(int board_des[64], int board_src[64]);
void display(int bd[64]);
//void printboard(int bd[64],FILE* tfile);

int evaluate()
{
	int i;
	int total=0;
	int lpiece;
	int lcolor;

	for(i=0;i<64;i++)
	{

		lpiece = tboard[i] & MASK_PIECE;
		lcolor = tboard[i] & MASK_COLOR;
		if(lcolor == COLOR_WHITE)
		{
			total = total + pieces_value[lpiece];
		}else
		{
			total = total - pieces_value[lpiece];
		}
	}
	return total;
}

int evaluatebd(int bd[64])
{
	int i;
	int total=0;
	int lpiece;
	int lcolor;

	for(i=0;i<64;i++)
	{

		lpiece = bd[i] & MASK_PIECE;
		lcolor = bd[i] & MASK_COLOR;
		if(lcolor == COLOR_WHITE)
		{
			total = total + pieces_value[lpiece];
		}else
		{
			total = total - pieces_value[lpiece];
		}
	}
	return total;
}

/*
	this is our monster!!
*/
int maxi( int depth, bool maximize, int start, int stop) {
	int max;		//max or min
	int i;			//scan board
	int n_move;		//number of move
	int score;		//score for minimax
	int lpiece;		//piece identification
	int lcolor;		//color of piece
	int des_move;		//destination move
	int lbestpos;
	int lbestmove;

    if ( depth <= 0 ) {
		return evaluate();
```

```
            }

        if ( maximize )
        {
            //printf("depth %d\n ",depth);
         max = -V_MAX;

                copy_board( history_board[depth], tboard);
                //record history (of three kingdom)
                for ( i=start; i<stop; i++ ) {      //scan board
                        lpiece = tboard[i] & MASK_PIECE;
                        lcolor = tboard[i] & MASK_COLOR;
                        if(( lpiece != EMPTY) && ( lcolor == COLOR_WHITE))
                        {
                                if(lpiece==PAWN)
                                {
                                        for ( n_move = 0; n_move < 3; n_move++ )
                                        {
                                                des_move = movement[lpiece][n_move];
                                /* des_move = -movement[lpiece][n_move]; if its BLACK */
                                                if(check_pawn_move(i,des_move))
                                                {
                                                        move_pseudo_pawn(i,des_move);
                                                        score = maxi( depth - 1, false, 0,
64 );
                                                        if(score>max)
                                                                {
                                                                        max=score;

                                                                }
                                                        copy_board(tboard,
history_board[depth]);
                                                }
                                        }

                                }else
                                {
                                        for ( n_move = 0; n_move <
number_of_move[lpiece]; n_move++ )
                                        {
                                                des_move=0;
                                                do
                                                {

        des_move=des_move+movement[lpiece][n_move];
                                                if(!check_move(i,des_move))
                                                {
                                                        break;
                                                }

                                                if(check_target_color(i,des_move))
        //if capture
                                                {

        move_pseudo_piece(i,des_move);
                                                        score = maxi( depth - 1,
false, 0, 64 );
                                                        if(score>max)
                                                        {
                                                                max=score;


                                                        }
                                                        copy_board(tboard,
history_board[depth]);        //return board
                                                        break; //stop
                                                }
```

```
                        //if empty
                                        move_pseudo_piece(i,des_move);
                                        score = maxi( depth - 1, false, 0,
64 );
                                        if(score>max)
                                                {
                                                        max=score;
                                                }
                                        copy_board(tboard,
history_board[depth]);          //return board

                                        }while(slide[lpiece]);

                                }
                                //castling move white
                                /*

      if((lpiece==KING)&&(bWhiteAllowCastlingL)&&(bWhiteAllowCastlingR))
                                        {
                                                for()
                                                {

                                                }
                                        }
                                        */
                                }
                        }
                }

                //printf("\nstart pos: %d destination: %d\n
",start_pos_move,des_pos_move);
        //printf("value %d \n ",max);
                return max;
        }else   //minimum
        {
            //printf("depth %d\n ",depth);
         max = V_MAX;

                copy_board( history_board[depth], tboard);
                //record history (of three kingdom)
                for ( i=0; i<64; i++ ) {            //scan board
                        lpiece = tboard[i] & MASK_PIECE;
                        lcolor = tboard[i] & MASK_COLOR;
                        if(( lpiece != EMPTY) && ( lcolor == COLOR_BLACK))
                        {
                                if(lpiece==PAWN)
                                {
                                        for ( n_move = 0; n_move < 3; n_move++ )
                                        {
                                                des_move = -movement[lpiece][n_move];
                                /* des_move = -movement[lpiece][n_move]; if its BLACK */
                                                if(check_pawn_move(i,des_move))
                                                {
                                                        move_pseudo_pawn(i,des_move);
                                                        score = maxi( depth - 1, true, 0,
64 );
                                                        if(score<max)
                                                                {
                                                                        max=score;
                                                                        lbestpos=i;
                                                                        lbestmove=des_move;
                                                                }
                                                        copy_board(tboard,
history_board[depth]);
                                                }
```

```
                        }

                    }else
                    {
                        for ( n_move = 0; n_move <
number_of_move[lpiece]; n_move++ )
                        {
                            des_move=0;
                            do
                            {

      des_move=des_move+movement[lpiece][n_move];
                                if(!check_move(i,des_move))
                                {
                                    break;
                                }

                                if(check_target_color(i,des_move))
      //if capture
                                {

      move_pseudo_piece(i,des_move);
                                    score = maxi( depth - 1,
true, 0, 64 );

                                    if(score<max)
                                    {
                                        max=score;
                                        lbestpos=i;
                                        lbestmove=des_move;
                                    }
                                    copy_board(tboard,
history_board[depth]);      //return board

                                    break;          //stop
                                }

                                move_pseudo_piece(i,des_move);
                                score = maxi( depth - 1, true, 0,
64 );

                                if(score<max)
                                    {
                                        max=score;
                                        lbestpos=i;
                                        lbestmove=des_move;
                                    }
                                copy_board(tboard,
history_board[depth]);          //return board


                            }while(slide[lpiece]);

                        }
                        //castling move white
                        /*

      if((lpiece==KING)&&(bWhiteAllowCastlingL)&&(bWhiteAllowCastlingR))
                        {
                            for()
                            {

                            }
                        }
                        */
                    }
                }
            }
```

```
		start_pos_move= lbestpos;
		des_pos_move= lbestmove;
			value_pos_move= max;
			//printf("\nstart pos: %d destination: %d\n
",start_pos_move,des_pos_move);
		//printf("value %d \n ",max);
			return max;
		}
}
 /**
		check_move mechanism:
		1. check if out of bound
		2. check if empty
		3. check friendly piece

 */
bool check_move(int piece_pos, int des_move)
{
		int lpiece=board[piece_pos] & MASK_PIECE;
		int lcolor=board[piece_pos] & MASK_COLOR;
		int lcolor_target;
		int lpiece_target;

		int mb_pos=mailbox64[piece_pos];		//convert board into mailbox
		int destination = mailbox[mb_pos+des_move];
		//if destination = -1 its out of bound, otherwise board array index

		//1. check boundary
		if(destination==-1)
		{
			//out of bound
			return false;
		}else{
			lcolor_target = tboard[destination] & MASK_COLOR;
			lpiece_target = tboard[destination] & MASK_PIECE;
			if((lpiece_target) == EMPTY) //2. check if empty
			{
				return true;
			}else				//3. if not empty, check their color
			{

				if(lcolor==lcolor_target)	//you cannot attack same color
				{
					//hold your attack, its our ally
					return false;
				}else
				{
					//yes, kill him!!
					return true;
				}
			}
		}
}

//check if its empty and have different color, return true if they have different
color
bool check_target_color(int piece_pos, int des_move)
{
		int lpiece=tboard[piece_pos] & MASK_PIECE;
		int lcolor=tboard[piece_pos] & MASK_COLOR;
		int lpiece_target;
		int lcolor_target;

		int mb_pos=mailbox64[piece_pos];		//convert board into mailbox
		int destination = mailbox[mb_pos+des_move];
		//if destination = -1 its out of bound, otherwise board array index
```

```
        if(destination==-1)
        {
                //out of bound,return -1
                return false;
        }else
        {
                lpiece_target=tboard[destination] & MASK_PIECE;
                lcolor_target=tboard[destination] & MASK_COLOR;
                if(lpiece_target==EMPTY)
                {
                        return false;
                }else
                {
                        if(lcolor==lcolor_target)
                        {
                                return false;
                        }else
                        {
                                return true;
                        }
                }
        }
}

bool check_pawn_move(int piece_pos, int des_move)
{
        int lpiece=board[piece_pos] & MASK_PIECE;
        int lcolor=board[piece_pos] & MASK_COLOR;
        int lcolor_target;
        int lpiece_target;

        int mb_pos=mailbox64[piece_pos];            //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index

        //1. check boundary
        if(destination==-1)
        {
                //out of bound
                return false;
        }else{
                lcolor_target=tboard[destination] & MASK_COLOR;
                lpiece_target=tboard[destination] & MASK_PIECE;
                if((des_move==10)||(des_move==-10))
                {
                        if(lpiece_target == 0)      //2. check if empty
                        {
                                return true;
                        }else
                        {
                                return false;
                        }
                }
                if((des_move==9)||(des_move==-9)||(des_move==11)||(des_move==-11))
                {
                        if( (lpiece_target != 0) && (lcolor_target!=lcolor))
                        //2. check if empty
                        {
                                return true;
                        }else
                        {
                                return false;
                        }
                }
        }
```

```
}

//exclusive for player
int move_player_piece(int piece_pos, int des_move)
{
    int piece_type=board[piece_pos]&MASK_PIECE;
    board[des_move]=board[piece_pos];
       board[piece_pos]=EMPTY;

       //promotion
       if(piece_type==PAWN)
    {
        if(des_move>=56)
        {
            board[des_move]=QUEEN;
        }
    }
    if(piece_type==ROOK)
    {
        if(piece_pos==0)
            {bWhiteAllowCastlingL=false;}
        if(piece_pos==7)
            {bWhiteAllowCastlingR=false;}
    }
    if(piece_type==KING)
    {
        bWhiteAllowCastlingL=false;
        bWhiteAllowCastlingR=false;
        //castling left
        if(des_move==C1)
        {
            board[A1]=EMPTY;
            board[C1]=KING;
            board[D1]=ROOK;
        }
        //castling right
        if(des_move==G1)
        {
            board[H1]=EMPTY;
            board[G1]=KING;
            board[F1]=ROOK;
        }
    }
    return 0;
}

bool check_move_player(int player_start, int player_des)
{
       int i;
       bool m_match = false;
       int lpiece = board[player_start] & MASK_PIECE;
       int lcolor = board[player_start] & MASK_COLOR;
       int lcolor_target;
       int lpiece_target;
       int mb_pos = mailbox64[player_start];      //convert board into mailbox
       int destination = 0;
       //if destination = -1 its out of bound, otherwise board array index
       int    startDesDiff = 0;
       int checkDesMove;
       int allowedMove[32];
       int allowedMoveCount=0;

       //check piece
       if(lpiece==EMPTY)
              { return false;}
```

```
if(lcolor!=COLOR_WHITE)
        { return false;}

//check movement rule
if(lpiece==PAWN)
{
 //only allow movement if player_des - player_start= 7/8/9
 startDesDiff=player_des-player_start;
 lpiece_target=board[player_des] & MASK_PIECE;
 lcolor_target=board[player_des] & MASK_COLOR;
 switch(startDesDiff)
 {
 case 8:
     if(lpiece_target==EMPTY)
     {
         return true;
     }
     break;
 case 7:
     if(lcolor_target==COLOR_BLACK)
     {
         return true;
     }else
     {
         return false;
     }
     break;
 case 9:
     if(lcolor_target==COLOR_BLACK)
     {
         return true;
     }else
     {
         return false;
     }
     break;
 default:
     return false;
     break;
 };

}
else
{
 for(i=0;i<number_of_move[lpiece];i++)
 {
         checkDesMove=0;
         do
         {
             checkDesMove = checkDesMove+movement[lpiece][i];
             destination = mailbox[mb_pos+checkDesMove];
             //out of bound
             if(destination==-1)
             {
                 break;
             }

             lpiece_target=board[destination] & MASK_PIECE;
             lcolor_target=board[destination] & MASK_COLOR;

             //friendly fire
             if((lpiece_target!=EMPTY)&&(lcolor_target==COLOR_WHITE))
             {
                 break;
             }
```

```
                    if((lpiece_target!=EMPTY)&&(lcolor_target==COLOR_BLACK))
                    {
                        allowedMove[allowedMoveCount]=destination;
                        allowedMoveCount++;
                        break;
                    }

                    if(lpiece_target==EMPTY)
                    {
                        allowedMove[allowedMoveCount]=destination;
                        allowedMoveCount++;
                    }

                }while(slide[lpiece]);
            }
            if(allowedMoveCount>0)
            {
                for(i=0;i<allowedMoveCount;i++)
                {
                    if(allowedMove[i]==player_des)
                    {
                        m_match=true;
                    }
                }
            }
            //castling check
            if(lpiece==KING)
            {
                if((player_des==C1)&&(bWhiteAllowCastlingL)&&(board[A1]==ROOK))
                {
                    if((board[B1]==EMPTY)&&(board[C1]==EMPTY)&&(board[D1]==EMPTY))
                    {
                        return true;
                    }
                }
                if((player_des==G1)&&(bWhiteAllowCastlingR)&&(board[H1]==ROOK))
                {
                    if((board[F1]==EMPTY)&&(board[G1]==EMPTY))
                    {
                        return true;
                    }
                }
            }
        }
        return m_match;

}

//return destination array
int move_pseudo_piece(int piece_pos, int des_move)
{
        int tpiece=tboard[piece_pos];
        int mb_pos=mailbox64[piece_pos];            //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index


    //printf("\npos+move %d\ndestin %d\n ",mb_pos+des_move,destination);
        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
                tboard[destination]=tpiece;
                tboard[piece_pos]=EMPTY;
```

```
//      display(tboard);
//      printf("\nmove %d to %d\n ",piece_pos,destination);
//      printboard(tboard,fftest);

                return destination;
        }
}

//special move for pawn (including queen promotion)
int move_pseudo_pawn(int piece_pos, int des_move)
{
        int tpiece=tboard[piece_pos];
        int lcolor=board[piece_pos] & MASK_COLOR;
        int mb_pos=mailbox64[piece_pos];            //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index

        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
                tboard[piece_pos]=EMPTY;
                tboard[destination]=tpiece;
                if(lcolor==COLOR_BLACK)
                {
                        if(destination<8)
                        {
                                tboard[destination]=QUEEN+COLOR_BLACK;
                                //color white=0, so queen+color white=5+0 :P
                        }
                }
                if(lcolor==COLOR_WHITE)
                {
                        if(destination>55)
                        {
                                tboard[destination]=QUEEN+COLOR_WHITE;
                                //color white=0, so queen+color white=5+0 :P
                        }
                }
//      display(tboard);
//      printf("\nmove %d to %d\n ",piece_pos,destination);
//      printboard(tboard,fftest);
                return destination;
        }
}


//return destination array, its black move/AI move only
int move_piece(int piece_pos, int des_move)
{
        int tpiece=board[piece_pos];
        int piece_type=board[piece_pos] & MASK_PIECE;
        int mb_pos=mailbox64[piece_pos];            //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index

        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
                board[destination]=tpiece;
                board[piece_pos]=EMPTY;
                //promotion for black
                if((piece_type==PAWN)&&(destination<8))
```

```
        {
            board[destination]=COLOR_BLACK+QUEEN;
        }
            return destination;
        }
}

//special move for pawn (including queen promotion)
int move_pawn(int piece_pos, int des_move)
{
        int tpiece=board[piece_pos];
        int lcolor=board[piece_pos] & MASK_COLOR;
        int mb_pos=mailbox64[piece_pos];          //convert board into mailbox
        int destination = mailbox[mb_pos+des_move];
        //if destination = -1 its out of bound, otherwise board array index

        if(destination==-1)
        {
                //out of bound
                return -1;
        }else{
            board[destination]=tpiece;
                board[piece_pos]=EMPTY;
                if(lcolor==COLOR_BLACK)
                {
                        if(destination<8)
                        {
                                board[destination]=QUEEN+COLOR_BLACK;
                                //color white=0, so queen+color white=5+0 :P
                        }
                }
                return destination;
        }
}

int find_max(int value_a, int value_b)
{
        if(value_a>value_b)
        {
                return value_a;
        }else
        {
                return value_b;
        }
}

int find_min(int value_a, int value_b)
{
        if(value_a<=value_b)
        {
                return value_a;
        }else
        {
                return value_b;
        }
}

void copy_board(int board_des[64], int board_src[64])
{
        int i;
        for(i=0;i<64;i++)
        {
                board_des[i]=board_src[i];
        }
}
```

```
void display(int bd[64])
{
        int i;
        printf("\n\n ");
        for(i=0;i<64;i++)
        {
            if(bd[i]==0)
            {
                printf(" .");
            }else
            {
                printf(" %c",bd[i]+'a');
            }
               if ((i + 1) % 8 == 0 && i != 63)
                       printf("\n ");
        }
        printf("\n");
}

void printboard(int bd[64],FILE* tfile)
{
        int i;
        fprintf(tfile," ");
        for(i=0;i<64;i++)
        {
            if(bd[i]==0)
            {
                fprintf(tfile," .");
            }else
            {
                fprintf(tfile," %c",bd[i]+'a');
            }
               if ((i + 1) % 8 == 0 && i != 63)
                       fprintf(tfile,"\n ");
        }
        fprintf(tfile,"\n");
        fprintf(tfile,"temporary value %d\n\n",evaluate());
}

int check_winning_player()
{
    int i;
    int winner=0;

    for(i=0;i<64;i++)
    {
        if(board[i]==KING)  winner++;
        if(board[i]==(COLOR_BLACK+KING))  winner--;
    }
    return winner;
}

int check_input(char* ch)
{
    int tpos=-1, tdes=-1;
    if((ch[0]<'a')||(ch[0]>'h')) return -1;
    if((ch[1]<'1')||(ch[1]>'8')) return -1;
    if((ch[2]<'a')||(ch[2]>'h')) return -1;
    if((ch[3]<'1')||(ch[3]>'8')) return -1;

    tpos=ch[0]-'a';
    tpos=tpos+(ch[1]-'1')*8;
    tdes=ch[2]-'a';
    tdes=tdes+(ch[3]-'1')*8;
    return (tpos<<6)|tdes;
}
```

```
//its just dumb main function
int main(int argc, char **argv){
	// MPI inits
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &node_count);
	MPI_Comm_rank(MPI_COMM_WORLD, &node_rank);
	MPI_Get_processor_name(node_name, &node_namelen);

	int i, j, k;
	int max, current;
	start_pos_move_global= malloc(sizeof(int)*node_count);
	des_pos_move_global= malloc(sizeof(int)*node_count);
	value_pos_move_global= malloc(sizeof(int)*node_count);

	i_start= 64/node_count*node_rank;
	i_stop = 64/node_count*(node_rank+1);

    char c1[16],c2;
    int tpos,tdes;
    int input_return;
    int winner=0;
    bool gameover=false;
    //fftest=fopen("../filetest.txt","w");
    //if(fftest==NULL) return -1;

    bWhiteAllowCastlingL=true;
	bWhiteAllowCastlingR=true;
	copy_board(board, init_board);

	while(!gameover)
	{
		if(ply%2==0)//white one
		{
			if (node_rank==0){
				// masukan dari sini
				display(board);
				//printf("Input your move: ");
				scanf("%s",&c1);
				input_return=check_input(c1);
				if(input_return<0) printf("\nWrong input, try again\n");
				else
				{
					tpos=input_return>>6;
					tdes=input_return&63;
					if(check_move_player(tpos,tdes))
					{
						move_player_piece(tpos,tdes);
						ply++;
						// boardcast board
						for (i=0; i<node_count; i++){

	MPI_Send(&board,64,MPI_INT,i,TAG_BOARD,MPI_COMM_WORLD);
						}
//						display(board);
						printf("\nPlayer moves %d to
%d\n",tpos,tdes);

					}
					else printf("\nOops!! illegal move, try
again\n");
				}
			}
			else {
				// recv board from broadcast
```

```
        MPI_Recv(&board,64,MPI_INT,0,TAG_BOARD,MPI_COMM_WORLD,&status);
                        ply++;
                        }
                }

                else  //computer's turn
                {
                    copy_board(tboard,board);
                        gettimeofday(&time_start_global,NULL);
                gettimeofday(&time_start,NULL);
                if (ply<3){ // opening book
                        start_pos_move= 52;
                        des_pos_move= -10;
                        value_pos_move= 500;
                        }
                        else maxi(DEPTH, false, i_start, i_stop);
                        gettimeofday(&time_stop,NULL);

                printf("\nNode%d suggested (%d to %d) val=%d, t=%.0f usec",node_rank,
                        start_pos_move, des_pos_move, value_pos_move,

(time_stop.tv_sec-time_start.tv_sec)*1e6+(time_stop.tv_usec-time_start.tv_usec));
                // send pos_moves and val to master
                MPI_Send(&start_pos_move, 1, MPI_INT, 0, TAG_POS_STR, MPI_COMM_WORLD);
                MPI_Send(&des_pos_move, 1, MPI_INT, 0, TAG_POS_DES, MPI_COMM_WORLD);
                MPI_Send(&value_pos_move, 1, MPI_INT, 0, TAG_VAL, MPI_COMM_WORLD);

                        current= 0;
                        if (node_rank==0){
                                // master select from the best val
                                for (i=0; i<node_count; i++){
                                // recv start_pos, des_pos, val from underlings

MPI_Recv(&(des_pos_move_global[i]),1,MPI_INT,i,TAG_POS_DES,MPI_COMM_WORLD,&status);

MPI_Recv(&(start_pos_move_global[i]),1,MPI_INT,i,TAG_POS_STR,MPI_COMM_WORLD,&status);

MPI_Recv(&(value_pos_move_global[i]),1,MPI_INT,i,TAG_VAL,MPI_COMM_WORLD,&status);
                                        if (value_pos_move_global[i]>current)
                                                max= i;
                                                current= value_pos_move_global[i];
                                        }

                        move_piece(start_pos_move_global[max], des_pos_move_global[max]);
                                gettimeofday(&time_stop_global,NULL);
                                //display(board);

printf("\nExecuted move: %d %d",start_pos_move_global[max],des_pos_move_global[max]);
                                //printf("\nCurrent score: %d",evaluate());

printf("\nTotal processing time: %.0f usec",

(time_stop_global.tv_sec-time_start_global.tv_sec)*1e6\

+(time_stop_global.tv_usec-time_start_global.tv_usec));
                                // broadcast board
                                for (i=0; i<node_count; i++)
                                {

        MPI_Send(&board,64,MPI_INT,i,TAG_BOARD,MPI_COMM_WORLD);
                                }
                        }
                        // recv recently boardcasted board
                else MPI_Recv(&board,64,MPI_INT,0,TAG_BOARD,MPI_COMM_WORLD,&status);
                        ply++;
```

```
                }


                winner=check_winning_player();
                if(winner!=0)
            {
                gameover=true;
                if(winner==1)
                {
                    printf("\n\nwinner is WHITE\n");
                }else
                {
                    printf("\n\nwinner is BLACK\n");
                }
            }
           }
           //fclose(fftest);
           MPI_Finalize();
           return(0);
}

/* //from wiki-wikipedia
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := -∞
        for each child of node
            val := minimax(child, depth - 1, FALSE)
            bestValue := max(bestValue, val)
        return bestValue
    else
        bestValue := +∞
        for each child of node
            val := minimax(child, depth - 1, TRUE)
            bestValue := min(bestValue, val)
        return bestValue
*/
```